The Report Committee for Kenneth Walter Fiduk, III
Certifies that this is the approved version of the following report:

What Can The .NET RDBMS Developer Do?
A Brief Survey of Impedance Mismatch Solutions For The .NET Developer

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Christine Julien

Sarfraz Khurshid

What Can The .NET RDBMS Developer Do?
A Brief Survey of Impedance Mismatch Solutions For The .NET Developer

by

Kenneth Walter Fiduk, III, BSCS

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2009

Abstract

What Can The .NET RDBMS Developer Do?
A Brief Survey of Impedance Mismatch Solutions For The .NET Developer

Kenneth Walter Fiduk, III, MSE

The University of Texas at Austin, 2009

Supervisor: Christine Julien

Nearly all modern software applications, from the simplest website user account system to the most complex, enterprise-level, completely-integrated infrastructure, utilize some sort of backend data storage and business logic that interacts with the backend. The ubiquitous nature of this backend/business dichotomy makes sense as the need to both store and manipulate data can be traced as far back as the Turing Machine in Computer Science. The most commonly used technologies for these two aspects are Relational Database Management Systems (RDBMS) for backend and Object-Oriented Programming (OOP) for business logic. However, these two methodologies are not immediately compatible and the inherent differences between data represented in RDBMS and data represented in OOP are not trivial.

Taking a .NET developer's perspective, this report aims to explore the RDBMS/OO dichotomy and its inherent issues. Schema management theory and algebra are discussed to gain better perspective of the domain and a survey of existing solutions for the .NET environment is explored. Additionally, methods outside the mainstream are discussed. The advantages and disadvantages of each are weighed and presented to the reader to help aid in design implementations in the future.

Table of Contents

List of Figures

1. Introduction

The promise and allure of computers for years has been their ability to remove the burden of mundane, repetitive tasks from a human and happily take them upon themselves. Numerous onerous tasks are now the responsibility of computers and software applications, mostly producing output faster and with fewer errors than their previous human counterparts. Great gains in efficiency can be had by implementing a system that handles predictable, repetitive, and deterministic tasks. This undertaking is often achieved by recreating the task, all participating actors, and the way they interact in some way within the system to enable virtual execution.

Mundane, repetitive tasks are seen frequently in code itself as well and they too are excellent candidates for delegation to automated processes. In today's modern desktop and web applications, both consumer and commercial, the task of manipulating and persisting data is nearly ubiquitous, often times the primary focus of the application. While the specific nature of the data is highly variable and dependent on the application itself, the underlying need for persisting the data is common and similar across the board.

This common need is not trivial. The vast majority of databases, where the data is persisted, are Relational Database Management Systems. The vast majority of applications, where the data is manipulated and presented, are written in Object-Oriented programming languages. The data, while meaning the same thing, are represented very differently in the two spheres. Translating from a relational domain to an object-oriented domain (and vice versa) is a burden that must be accepted and carried by the application, regardless of the application logic driving the data.

This paper aims to explore this common problem, also called "impedance mismatch", and discuss some of the many ways it can be properly addressed. A brief

discussion of database theory and model management is used to help define and understand the problem and problem space. The current state of most software applications is then discussed where we look at some of the problems, and their consequences, that arise from the mismatch. Next, a look at a class of solutions popularly used today to alleviate impedance mismatch, Object-Relational Mappings (ORMs), and a handful of specific ORM products are addressed. Finally, potential future and forward-thinking solutions are discussed. The reader will have a good understanding of the problem of working across a relational-object boundary and an idea of the many available mitigation strategies by the end of the paper.

I take a .NET-centric approach for this paper for many reasons. First and foremost, the majority of my career has been spent developing and maintaining .NET (specifically C#) applications and thus first-hand experience of working with impedance mismatch in .NET is plentiful. Also, the .NET framework enjoys wide adoption across the industry in regards to commercial or line-of-business application development. This paper is meant as a practical guide to handling the RDBMS/OO data model disconnect and discussing real-world solutions for a common real-world environment is a priority. Focusing on .NET helps provide this practical element.

2. Background

Fortunately, we can build upon the multitudes of research found in the field of database theory and management. There are ways to alleviate the pain of writing applications that must overcome the impedance mismatch – the differing representations of data between the relational database and the object-oriented programming language.

The problem can be addressed directly, by way of frameworks and code generators that handle the manual labor of plumbing code themselves. It can also be removed from the table entirely by removing the relational database from the situation altogether. The entire situation can be turned upside down on its head by rearranging the classical roles of data model definition in an OOP application across the board.

We will begin by briefly discussing what data modeling is, why we need it, and what can be done with it. By exploring the theory behind data modeling, we can better assess what can be done and what is available in our tool bag as software developers to alleviate development, maintenance, product specification issues and the friction that comes with them.

2.1. Model Management

A proper introduction to relational databases and its underlying theory is outside the scope of the paper, but it is worth acknowledging such foundational work when looking at different ways of surfacing and manipulating the data therein. Typically, information is stored as tuples (rows) of a relation (table) in your modern relational database management system (RDBMS). Dependencies and relationships are established between such relations and, collectively, are referred to as a schema. Relations are given a name and a list of associating attributes (columns). Associations between relations are

established typically through foreign key references – referencing another relation's "primary" key in your own. Constraints can be established on these associations, from simple existence tests to sophisticated custom logic established by the database administrator. This ability to define data and its associations is what makes the RDBMS such a predominant beast [17].

It can be, and has been, argued that any system of sufficient size and/or complexity should be ruled by its data model. Business logic, workflow, and behavior are often defined only in terms of the domain model, be it bank transactions (customers, accounts, cash, etc.), medical records (people, diagnoses, drugs, etc.), website design (users, posts, comments, articles), or even selling a fast food hamburger (hungry person, food, restaurant, etc.). Processes are usually data-centric and data-driven. The ability to accurately define both models and their dependencies in an RDBMS is its selling point as a storage model.

Clearly, modeling is the basis of today's systems. It is important to note there is more than one system. One system's model is not the same as another system's model. Granted, this both makes sense and presents no problem in many cases; there is probably little need for the bank to have a "hamburger" entity in its domain! To continue an admittedly absurd analogy, consider the "restaurant" and "hungry person" from earlier as self-contained systems rather than simple models. One could easily imagine it might be pertinent for both systems to not only know what a "hamburger" is, but also have an agreed upon understanding of such a definition so that the process of selling a hamburger can properly occur.

Turning away from the tempting philosophical tangent ("What is the hamburger in-and-of-itself, really?"), the ability for one system to communicate with another with a common understanding is both important and ubiquitous. Disparate models and

mitigation strategies have been the subject of attention for decades now, with origins outside of software development. Complaints of low productivity and error-ridden results have been heard for years across all disciplines [12]. A large amount of research has been conducted regarding models and their ability to map between each other for rich communication [20] [22] [17] [7]. This concept is exactly what the subject matter of this paper is concerned with – the ability to seamlessly communicate between the database model and the object model in code.

Meta data management might be a more apt term seeing how it is more concerned with the description of data rather than the data itself.

Research in the area of Model Management provides the foundation for us to explore specific mapping technologies for the .NET developer. Figure 1 shows the conceptual structure of two models representing the same data and a mapping between them.

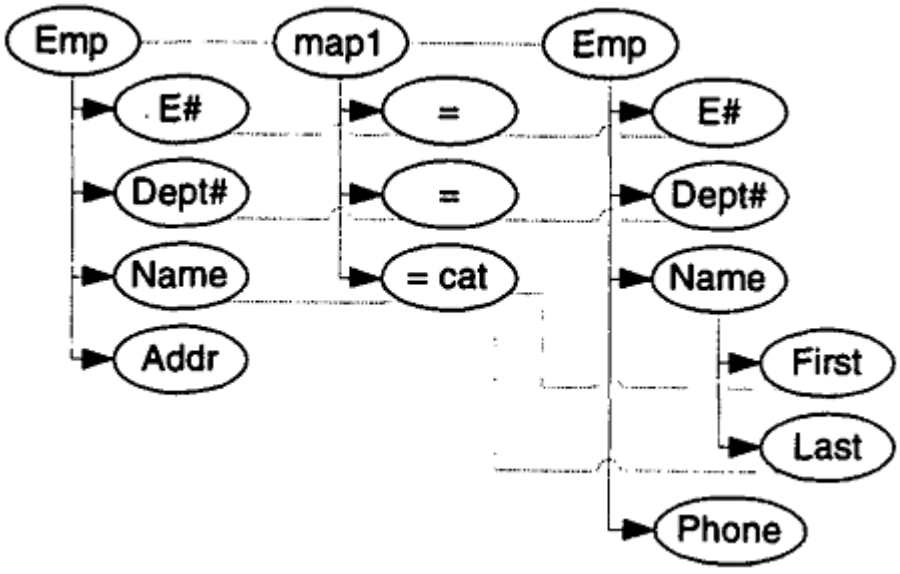


Figure 1 Sample Data Models and Mapping [7]

Mappings such as “map1” in Figure 1 can be manipulated themselves. It has been algebraically proven that basic functions like Match, Compose, Diff, Merge, etc. on mappings preserve all underlying models – their shapes remain intact throughout manipulation [17] [7]. This means that any combination of model mapping manipulation (such as the merging of disjoint mappings), provided it is done correctly, can be executed without breaking any definitions in the model [20] [3] [6]. This provides reassurance that model mapping can be done automatically, unmonitored, with confidence.

Why is this? Because the mapping duties can be taken out of the developer’s hands. Nearly all commercial programmers are paid to implement business logic in some form or another. Management wants results and rarely appreciates or acknowledges the required heavy lifting to get it done. Additionally, software engineers are not exactly cheap, so the more time spent on the problem at hand instead of creating trivial “plumbing” code, the more cost efficient the engineer is for the business. Model management research allows for great gains in programming correctness and productivity [7] [8].

Additionally, greater assurance in functional correctness can be gained when utilizing such mapping tools. A developer, despite all his or her expertise, will always introduce bugs during development. However, this is limited to the amount of code that is touched in the process. If a proven process, such as mapping between models, is successfully captured and implemented with code generation, then the developer has little opportunity to introduce bugs into the system.

Model management tools also open the possibilities for a developer by enhancing their toolbox and offering a wider, richer range of potential solutions to a given problem. This increased flexibility is valuable and widens the breadth of problems conquerable.

3. Today's Situation

Let us turn our direction towards more specific technologies and situations commonly encountered in today's software industry. As mentioned earlier, nearly all systems are data-centric, and the systems encountered in a typical line-of-business application are certainly no exceptions. The typical architecture seen is some variant of the three tiered system, consisting of a backend data storage tier, an application logic tier, and a presentation/view tier. Each tier is concerned with separate aspects of the same underlying data model. The backend is implemented using an RDBMS (usually wrapped in objects to expose an API), and the application and presentation tiers are written in an object-oriented language [16][1].

Popular technology stacks fitting this description include .NET (C#/VB.NET) and SQL Server or Java (J2EE) and Oracle. SQL Server and Oracle (along with other popular databases such as DB2, Informix, and MySQL) are relational databases, whereas the languages used to implement the application logic and viewing of the data are object-oriented programming languages [10]. This relational/object-oriented paradigm is exactly the situation described in the background section. Also referred to as impedance mismatch, this disconnect and its mitigation strategies is the motivation behind the paper.

At first glance, the two environments appear to share many similar traits, and this is not entirely incorrect. In its simplest form, a class definition in an object-oriented language (the simplest form implying an "anemic" model – one comprised solely of a collection of simple value-typed named attributes) is a direct analogue of the relation (table) definition found in database theory. Furthermore, rows in a table can be thought of as objects, or instances, of a class. Both capture the description of a data model in its own domain – for databases, its Database Definition Language (DDL) is used to define

the schema of a table, for OO, the programming language contains first-class concepts for defining a class.

However, the similarities fade away as we progress further down each technology path. As discussed previously, in addition to basic relation definitions, an RDBMS also supports referential integrity and constraints. Some systems even support table inheritance and a sense of data hierarchy. Classes in an OO language can contain references to other classes and internal logic and behavior. Both environments offer rich feature sets, with the two sharing a modest common set where the features overlap.

Despite these differences, they remain, in spirit, similar creatures. Once you remove traits of each that really do not have any analogous partner (i.e., tables have no concept of "behavior" – let alone higher level concepts like polymorphism and interface definitions, whereas classes have little concept of transactions or indexing as its focus is on behavior and process), what remains is the basic table-class relationship. This makes sense, or it should, as ultimately both domains are interested in representing, and communicating about, the same entity.

So, if the two domains describe the same data and share enough similar traits and concepts to draw apt comparisons, what is the big fuss? Why all this attention spent on bridging the not-too-distant gap between relational data and object-oriented data? How mismatched are the domains and can we limit the impedance? The gap can be easily crossed with some mapping code realizing the connection between the two and allowing for the exchanging of data, right?

Yes, it can, and buried within lies the problem. It DOES require mapping, and such mapping code is neither entirely trivial nor minimal. In fact, such code is often a common source of bugs when making changes to an existing system. A change in a property's data type, in either the database or code, will break the mapping code unless

otherwise properly addressed. This mapping code is often referred to as "plumbing" code, because it provides the necessary "plumbing" to correctly switch between the RDBMS and OO domains.

Recall in our model management discussion earlier that the basic components are models and their mappings, or in other words, the data in question and the steps necessary to represent the data in any given domain. Our plumbing code is exactly that: the mapping needed to translate data models in one domain, the RDBMS, to the other, OO programming, and vice versa. It is absolutely necessary for proper execution and storage of data in an RDBMS/OO environment [8].

Plumbing code is fragile, bloats code, and usually adds little to no business logic value. It can be repetitive, verbose, and disproportionately time consuming when measured against the lack of novel business value it presents to the problem at hand. The direct mapping of class properties to table columns and rows to objects is straight-forward and presents a minimal level of problem solving. However, classes can certainly be more complex, representing any sort of Is-A and Has-A relationship, composition, or other such combinations that do not necessarily have a direct analogue to the relational world. Such structures need to be explicitly handled in code for proper mapping to its RDBMS representation [26].

The need to transition between the two worlds usually centers on persisting changes made to the state of the data model. These changes take the form of a Create, Read, Update, or Delete function – more commonly, and endearingly, referred to as CRUD operations. Despite these operations all dealing with the same entity or model (i.e., creation, reading, updating, and deleting of an "Employee" entity), different code must be written for all four functions. Code is written for all four functions for every data

entity that requires persistence [26]. The same familiar adjectives come to mind when describing what effect this has on the code base: verbose, bloated, and fragile.

Without the help of tools or frameworks, such mapping code must be written and maintained by a software developer and must provide coverage for all objects intended for persistence. This can cause the maintenance code for each persisted object to greatly increase, resulting in verbose, bloated plumbing code across the system. More code means longer files and larger compiled libraries (in the case of .NET, assemblies), increasing the disk footprint of your application. While a dynamically-linked library possessing a couple hundred extra kilobytes may have little impact on the performance of the application (i.e., loading the library in main memory from disc takes longer the larger the file is), the more tangible and practical impact to larger files is simply an increased effort in navigation and comprehension of the code for the developer. A common, and virtuous, goal of the developer is to write succinct, easy-to-read, and easier-to-comprehend code, allowing for other developers to quickly situate themselves with the problem at hand. The more cryptic and cluttered the code, the greater the difficulty for other developers to work on the code, and collaboration is less effective.

Plumbing code also is a rich source of bugs during the development of an application. It may seem obvious, but the data model initially designed for a project is rarely the same model found at its completion. Redesigning and refactoring regularly occurs in all steps of the software development process, and any sort of change to the data model results in broken plumbing code. The property that once was an integer and now is a string or the renaming of an object generally requires alterations to its mapping between relational and OO domains. Sometimes bugs are easily caught and tracked down, but they can also be very illusive, especially if there is a lack of strong-typed support for the data. Objects, by virtue of being defined in the code itself, usually have

rich strong-typed support, but databases and their tables rarely, if at all, are strongly-typed when interacted with in code, but instead rely on string representations of column and table names, and are much more difficult in catching and preventing bugs when they are introduced. This fragile dependence on mapping code slows development productivity and increases debugging efforts.

In addition to producing bloated and brittle code, as mentioned before, writing and maintaining plumbing code is incredibly resource consuming. Rarely does the code possess any business or domain value; rather it usually is a means to achieve a desired end. Software developers spend years in school learning problem-solving skills, algorithms, and the inner workings of programs and their platforms of execution. The main asset they bring to the table when creating and developing an application is the ability to comprehend a business problem and design and implement a solution satisfying the requirements. Time spent by a developer on plumbing code is inefficient and misdirected. Developers are an expensive resource and their time is better spent on the business logic or problem at hand rather than mapping code.

Clearly, utilizing a software developer's time and skills to handwrite time-consuming code with little true business value in a highly bug-conducive coding environment is not the best way to go. If plumbing code is the implementation of the mappings between models as discussed in the section on database theory and model management, there must be a way to remove the burden of handwriting the code from software developers and instead leave the work to code generation tools and frameworks. As it turns out, this is exactly the case.

4. Hope Persists

Fear not, fair developers! In reality, there is a wide variety of tools and frameworks whose sole purpose is to address this very problem. Additionally, other less mature and exotic tools and methodologies exist that aim to address mapping between RDBMS and OOP in entirely different ways. The remainder of this paper presents several of these ideas, how each addresses the issues as discussed previously, and different pros and cons of each.

At this point, the author would like to remind the reader that this is a paper centered around the relational/OO disconnect for developers operating in a .NET environment, Microsoft's stack of programming and development tools. The vast majority of the author's career has been spent in this stack, and it is where most of his industry experience occurred. Additionally, keeping the scope of solutions to those implementable within .NET keeps the paper in a manageable state. Finally, the paper is intended as a practical guide for real solutions in industry rather than an esoteric, theoretical discussion.

4.1. Existing Tools, Frameworks – O/R Mapping

By far the biggest type of software in today's solutions for RDBMS/OOP development is called Object-Relational Mapping (also known as O/R Mapping software or ORM software). As can probably be discerned from its title, Object-Relational Mapping software specifically addresses the creation of the mapping ("plumbing") code needed to communicate between the relational domain and the OOP domain [22] [15] [26].

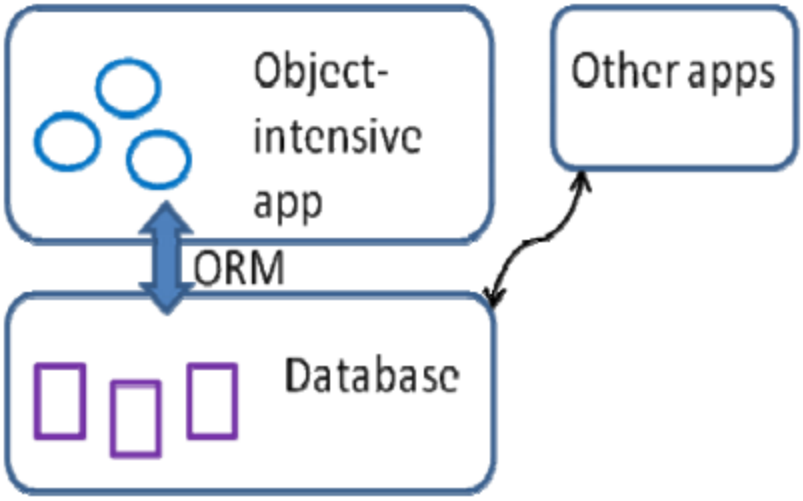


Figure 2 Basic ORM Architecture [21]

Their main concern is removing the need of developers to hand-write the code, instead utilizing the power of code generation to help automate the process. There are different approaches to this solution, which we will discuss later, ranging from writing code describing the mappings specifically at a higher level to actually defining the model schema itself within the tool so that the objects and databases fall out of the model. The tools that receive attention here specifically are Microsoft's ADO.NET Entity Framework (EF), the open-source framework NHibernate, and IBM's Rational Rose/UML utilities [21].

4.1.1. Microsoft ADO.NET Entity Framework

ADO.NET Entity Framework is a first-class Microsoft product in the .NET framework and as such, tends to fit in a .NET environment with the least resistance. EF builds on top of one of .NET's time-tested lower level database APIs, ADO.NET. ADO stands for ActiveX Data Object, and its origins actually predate the .NET framework, though ADO.NET is widely considered a rewrite of ADO [23].

Entity Framework, like the majority of ORMs available today, does not attempt to reorganize the core RDBMS/OOP paradigm to address the mapping but rather abstracts away the necessary code and logic from the developer. This product is an excellent

example of utilizing the known plumbing requirements and generating code to satisfy them. In general, generated code can be trusted to be correct, providing the underlying generation rules are sound, as a program, unlike a human, can be trusted to be consistent and accurate each and every time.

The greatest strength of Entity Framework, when compared to other similar tools, is its rich and powerful visual designer. Some may argue that a visual designer, by nature, is superficial and only serves to add additional layers of bloat for something that can be implemented in text files. Done poorly, this is certainly true and has been the fate of many visual design tools throughout the years. However, done well, a visual representation of an application's data models can be intuitive and make it easier to capture the larger essence of the model; a picture is worth a thousand words, or in this case, a diagram can be worth a thousand lines of code.

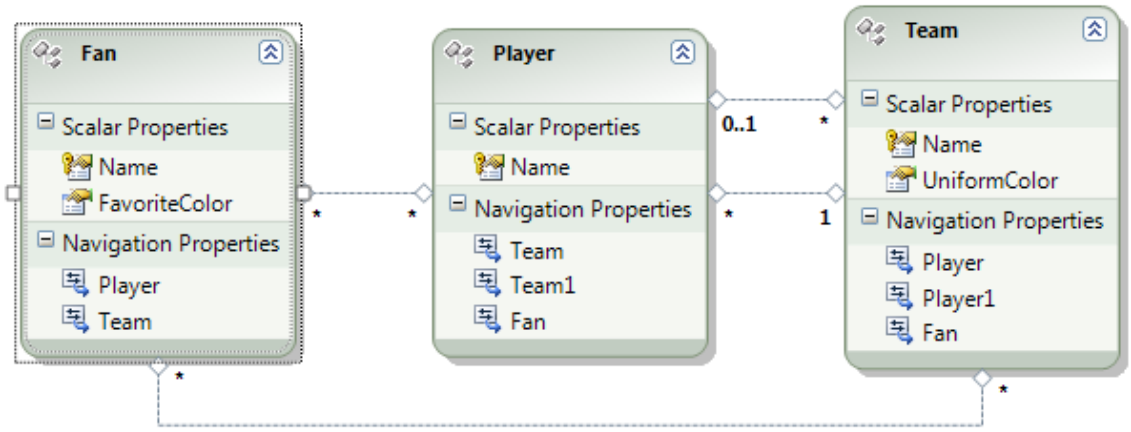


Figure 3 ADO.NET Entity Framework Designer Screenshot

Entity Framework supports all major database schema operations and constructs, including primary keys, foreign keys, constraints, and even stored procedures. The designer exposes these entities much like a database table diagram in a database

management application or an object graph in your favorite IDE – using rectangles to represent entities with a listing of properties and methods within them and associations represented as arrows and lines drawn between entities.

Strong database vendor support exists for Entity Framework with native .NET drivers available for databases like SQLServer, Oracle, DB2 (Informix, DB2, IBM Data Server, etc), and MySQL.

As mentioned earlier, Entity Framework simply hides the plumbing code from the developer and surfaces it with a straightforward GUI. The actual object-relational mapping code is found in the edmx file itself. Right-clicking on the file in the Solution Explorer of Visual Studio, selecting "Open with…", and choosing "XML Editor" reveals the actual XML of the data model. It is divided into three main sections: the storage model, the conceptual model, and the conceptual-storage mappings. These sound suspiciously similar to the database schema, object schema, and their translational mappings in model management jargon [2]. It will look something similar to Figure 4 below:

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0" xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
  <!-- EF Runtime content -->
  <edmx:Runtime>
    <!-- SSDL content -->
    <edmx:StorageModels>
      <Schema Namespace="EE382V_HW4_PartBModel.Store" Alias="Self" Provider="System.Data.Sql
    </edmx:StorageModels>
    <!-- CSDL content -->
    <edmx:ConceptualModels>
      <Schema Namespace="EE382V_HW4_PartBModel" Alias="Self" xmlns="http://schemas.microsoft
    </edmx:ConceptualModels>
    <!-- C-S mapping content -->
    <edmx:Mappings>
      <Mapping Space="C-S" xmlns="urn:schemas-microsoft-com:windows:storage:mapping:CS">
    </edmx:Mappings>
  </edmx:Runtime>
  <!-- EF Designer content (DO NOT EDIT MANUALLY BELOW HERE) -->
  <edmx:Designer xmlns="http://schemas.microsoft.com/ado/2007/06/edmx">
</edmx:Edmx>
```

Figure 4 Sample XML of the .edmx file for Entity Framework

The storage model is marked by the <edmx:StorageModels> tag and define the tables of the database, called EntitySets in the edmx. EntitySets are assigned names, EntityTypes, table names, and potentially defining queries. Foreign keys are represented by AssociationSets with Ends pointing to the Entities involved in the association. The table relations (EntityTypes) are then defined, enumerating out the columns (Property) and assigning primary keys.

The class schema is defined in a very similar manner in the ConceptualModels section. EntitySets, EntityTypes, AssociationSets, etc are all defined in a manner the object-oriented language can understand, using property names and native data types.

Having defined the database and class schemas for each domain, the edmx file then defines the mappings linking the two in the Mappings section. This section sees EntitySetMappings, EntityTypeMappings, and AssociationSetMappings defined by their appropriate ScalarProperties and EndProperties. Mapping names, in addition to conceptual names, can be edited to have custom names instead of the defaults provided by Entity Framework [2].

Link or join tables used in the database schema to represent many-to-many relationships are abstracted away by EF and instead are surfaced as their conceptual representation – many-to-many associations between two entities. This is done through the mapping of the many-to-many EntityType in the StorageModel to simple AssocationSets in the ConceptualModel.

The current version of Entity Framework (Ver 1, released with .NET 3.5 SP1) is hampered by a major weakness: the inability to generate database schema from a given data model. This will be addressed with the next version of EF, to be released around the release of .NET 4.0 and Visual Studio 2010. Termed “code-only”, database schema can be generated from a series of C# calls, given the existence of necessary database

providers. Version 1 has a dependence on valid database schema. As such, EF only surfaces database-centric concepts and results in a strained development experience. EF is an excellent step in the right direction, one strongly needed in the Microsoft stack of development tools, but proper expectations should be set when dealing with the first version.

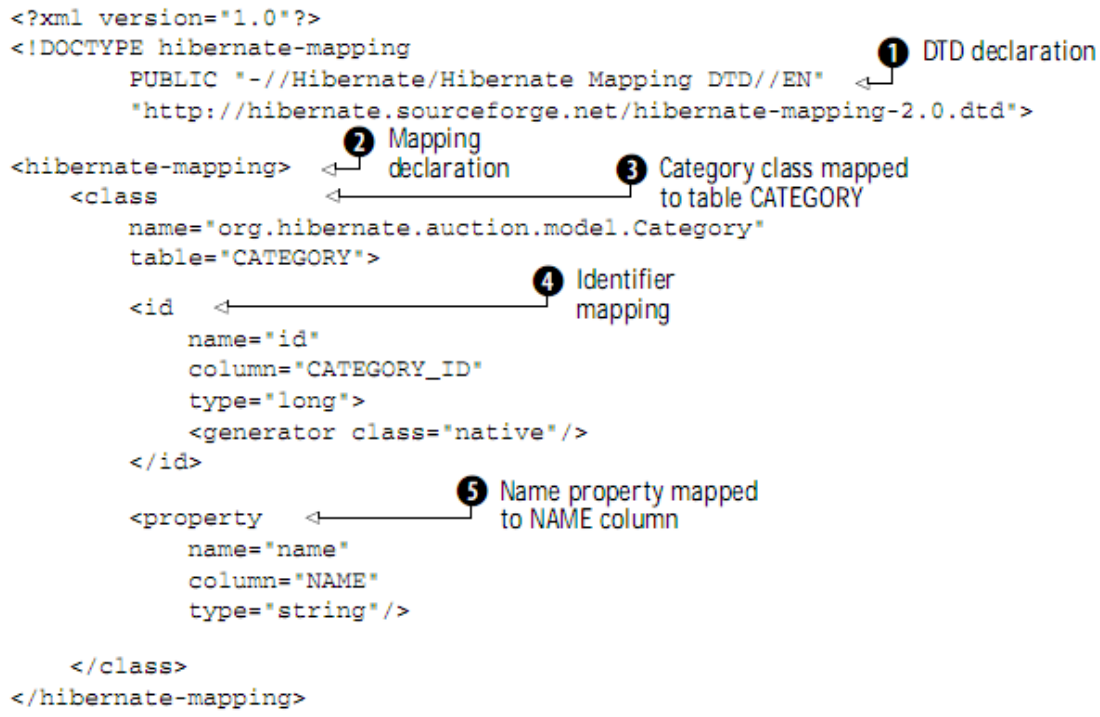
4.1.2. NHibernate

Another popular ORM used with .NET does not originate from Microsoft at all, but rather the open-source community surrounding Java. Unlike Entity Framework, NHibernate does not provide a visual designer. It instead offers a code-only solution for object-relational mapping. Make no mistake, though, the feature set offered by NHibernate actually surpasses that of EF and is done with a largely hands-off approach towards the entity classes in OOP. This ability alone makes it a more attractive option to a large number of developers when compared to Entity Framework.

Developers do not like having to markup their business objects with attributes or preprocessor instructions. It clutters code and makes business objects more difficult to read. Additionally, many developers prefer to have direct access to business object code. Entity Framework does not generate separate individual classes for each entity, the modus operandi of object-oriented programming, but rather encapsulates it all within its .edmx file. Where EF wraps everything in a proprietary edmx file, NHibernate instead hides nothing and utilizes simple, undecorated classes for its business entities. This can be a very attractive trait when working with legacy code whose business objects are already well established and mature.

NHibernate achieves this by way of defining mappings in relatively terse XML files. These files define the shapes of the business objects to be mapped. Outside of a

NHibernate configuration file used to define the database connection string, provider assemblies, and other options, nothing else is needed to begin to use NHibernate's ORM capabilities. Database schema can be generated based on these mapping XML files with NHibernate [5]. An example is found below in Figure 5:




```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
        PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
    <class
        name="org.hibernate.auction.model.Category"
        table="CATEGORY">

        <id
            name="id"
            column="CATEGORY_ID"
            type="long">
            <generator class="native"/>
        </id>

        <property
            name="name"
            column="NAME"
            type="string"/>

    </class>
</hibernate-mapping>
```

Figure 5 Sample NHibernate .mapping file [5]

Additionally, through the use of the repository and other patterns, NHibernate provides "persistence-ignorance" for the business objects. This simply means that the business objects are not responsible, nor care at all, for how they are persisted. This separation of concerns is a desirable trait for those working in large enterprise applications, for instance.

NHibernate also has strong support for transactions across maps of objects. Transactional integrity is a high level concern of concurrent database execution, and the concern surfaces in the object-oriented domain as well. NHibernate provides a robust

transaction management experience [21]. (Entity Framework also possesses strong transaction support, handling most common situations outside of the developer's hands.)

Last, and certainly not least, especially when appeasing management's budget demands, NHibernate is an open-source solution, available for use for any application of your choosing. Despite the absent price tag, NHibernate offers an incredibly mature (particularly in the ORM space) solution for the .NET environment.

4.1.3. Other ORMs

Several other popular ORM products, both commercial and open-source, can be found in the industry, including, but certainly not limited to: LLBLGen [http://www.llblgen.com/defaultgeneric.aspx], .netTiers [http://www.nettiers.com/], LightSpeed [http://www.mindscape.co.nz/products/LightSpeed/], and ActiveRecord [http://www.castleproject.org/activerecord/index.html]. This account is by no means meant to be a comprehensive one, but rather provide an idea of what the ORM-type solution has to offer and to highlight a couple of leading players in the field.

4.2. UML/Rational Rose

Rational Rose is the easily the most senior of the tools discussed in this paper and can be thought of as the swiss-army pocket knife of modeling needs. Firmly rooted in the Unified Modeling Language (UML), Rational Rose offers a powerful, if not entirely user-friendly, visual designer to define and describe models and their relationships. UML is a long standing effort to provide a standard modeling language and designer [19].

The vast majority of .NET development is done in the Microsoft Visual Studio IDE. Earlier versions saw support for Rational Rose/.NET interoperation by way of XDE Developer for Visual Studio, but Visual Studio 2008 no longer supports a UML tool to

the extent of XDE. Instead, IBM offers Rational Modeling Extension for Microsoft .NET within their Rational Software Architect suite [9].

Other UML tools exist for use with Visual Studio 2008, including Visual Paradigm SDE for Visual Studio [http://www.visual-paradigm.com/product/sde/vs/] and Sparx Enterprise Architect [http://www.sparxsystems.com.au/]. UML is a well-researched topic and there are multiple thoroughly documented resources available online for its structure and abilities. Such an exploration is both beyond the scope of the paper and beyond the knowledge (certainly the expertise) of the author [19].

5. Further, Promising Options

Solutions to the relational-OO disconnect, while certainly dominated by the approach today, are not limited to the Object-Relational Mapping approach. Intriguing developments in both technology and design can be found today that deserve attention as viable alternatives to ORM. Two such approaches are discussed here.

5.1. ODBMS – Object Database Management Systems

A solution that would completely side step impedance mismatch altogether is the use of Object-Oriented database persistence. Object-Oriented databases have their schema defined in object space rather than relational space, just like that of our programming language! No mapping is necessary when both the persistent layer and application layer represent the data in the exact same manner.

The concept of Object Database Management Systems arose in the mid to late 1980s and has experienced swells and reductions in popularity and perceived plausibility [18]. Today it has a firm hold in several data persistence needs and a lively open-source community continues to push contemporary ideas of the ODBMS and what it can offer. First the basic structure and requirements of an ODBMS will be described, followed by a discussion of its applications and plausibility in standard line-of-business type applications.

Today's perception and expectations of a viable ODBMS were largely shaped by Atkinson, et al. in the influential paper "The Object-Oriented Database System Manifesto [4]. In it, thirteen features are described as essential, five taken from the RDBMS domain and eight from the Object domain. The overall aim for an ODBMS is to provide the

referential and data integrity and utility of the modern RDBMS with the familiarity, convenience, and defining attributes of the modern Object-Oriented design.

The required behaviors from the RDBMS realm include: persistence, secondary storage management, concurrency, recovery, and ad hoc query facilities [4]. Persistence and secondary storage management simply mean that an ODBMS must be able to save and retrieve data on a disk, outside of a system's volatile memory. This is the primary service provided by a database system in general, regardless of data modeling methodology. Concurrency, recovery, and ad hoc query facilities constitute the second level of features expected of all modern RDBMS, and an ODBMS is no different. The ability to handle multiple database concurrent connections making complex commands involving foreign key dependencies, triggers, and constraints is crucial for the success of any database system. Interrogation of the data with a common query language is a demand of the ODBMS that presents unique challenges in comparison to its relational analogue. It also cannot be omitted, as the ability to query a database in an ad-hoc manner is and has been a staple of the software developer's toolkit. What would SQLServer be without SQLServer Query Analyzer?

In addition to what is drawn from the RDBMS realm, an ODBMS should possess the following object-oriented principles: complex object support, encapsulation, types/classes, inheritance, overriding with late binding, extensibility, and computational completeness [4]. The core tenets that make object-oriented design such a successful and popular programming methodology are encapsulation, inheritance, and polymorphism, and the eight attributes listed above are simply features that help provide these tenets. Persisted objects in an ODBMS should exhibit the same qualities as their counterparts in memory handled by code – class definitions, inheritance of properties and methods, polymorphism through late binding, and complex object maps with collections and

compositions across multiple object types. It is certainly foreign to imagine the persisted data model possessing abilities usually found in live computations. But therein lies the beauty of the approach: today's code works with objects, creating and manipulating representations of complex relations and data points, passing data across multiple application layers and tiers, often times in different execution environments; there should be no reason why the objects cannot simply persist as they are, rather than being forced into a foreign schema that may or may not offer a clean fit [13].

In an ODBMS, the class definition IS the schema. There are no table definitions as data is not represented as relations, but rather as objects. In an RDBMS, associations are explicitly defined with foreign key relationships, separate entities created to keep track of the entities involved in the association. In some cases, such as representing many-to-many associations between entities, a new table must be created to solely represent this association. Because the class definition is the schema in a pure object database, such relations are inherent in the schema definition and do not need to be explicitly defined. Mapping code to properly handle the CRUD operations of a composition or many-to-many relationship between two objects is completely avoided as the database natively understands the nature of their relationships. In fact, most ODBMS products will persist any and all objects residing in the object to be saved automatically (i.e.: the entire object map of the object to be saved) [13] [4].

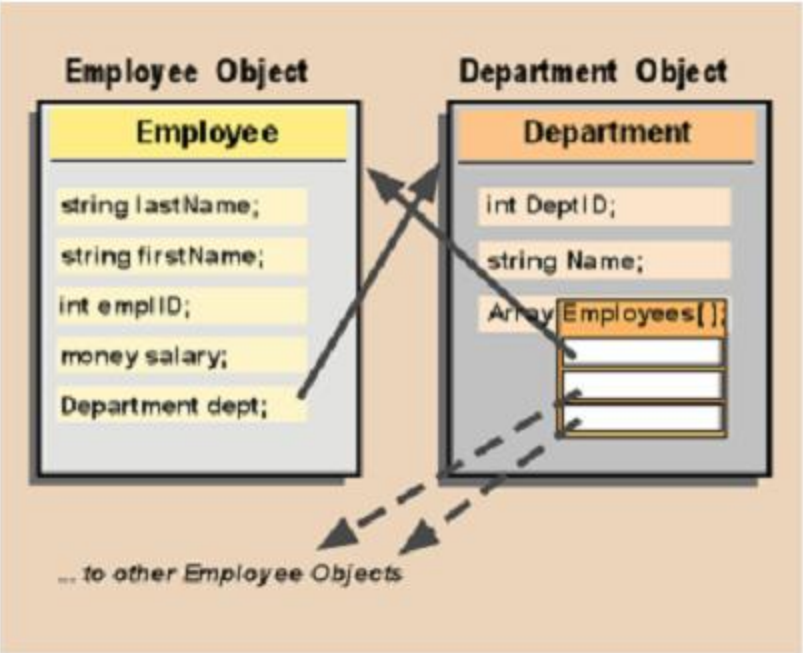


Figure 6 A Sample Object Database Schema [13]

Needless to say, a pure object database backend would greatly improve data access code. Less code is needed as no mapping code is required, maintainability increases as code is more intuitive and readable for future development, and quality and performance improve as execution responsibility is taken out of the developers' hands and given to the development platform, reducing the possibility of developer-introduced bugs.

So why has it not completely replaced the RDBMS as the go-to solution for persistence needs? There are many reasons, but it can be summed up nicely with the inability, thus far, of the ODBMS to provide a couple essential functions, particularly an ad hoc query facility, and the ability of the RDMBS to expose some ODMBS behavior with improved APIs and facilities. The popularity and usefulness of ORMs today have greatly improved the impedance mismatch, often times to the point that the developer completely forgets that he is working against an RDBMS backend. Additionally, the

introduction of the Object/Relational Database Management System (ORDBMS) has further extended the viability of relational databases. Like ORMs but integrated into the database system rather than in software that interacts with a traditional RDBMS, ORDBMS surfaces object operations from the database, extending the old feature set of the RDBMS. ORDBMS also provide additional data types for storage, handling audio and video (and other such media) files as objects in the database as relational storage traditionally has a difficult time saving such data [14] [18].

Another reason for the lack of widespread adoption of ODBMS is the lack of a complete implantation of a standard query language against object databases. A difficult road with promises and disappointments, several efforts have been made to create a SQL-analogue for the ODBMS over the years. Object Query Language (OQL) is a standard for querying object databases developed by the Object Data Management Group, ODMG, however vendors have thus far failed to reach full implementation, instead providing a subset of the standard in products. The OQL standard is rather complex in its entirety, but it still has helped pave the way to practical object querying languages such as working subsets of OQL, JDOQL (Java Data Object Query Language), and db4o SODA (Simple Object Database Access).

Recent work by William Cook has explored the idea of using Native Queries in object-oriented languages, leaving the querying capabilities to the languages themselves [11]. Since the paper in August 2005, additions to the C# language have provided a similar experience, if not a full realization, of what Cook possibly envisioned. The Language Integrated Query component of Microsoft's .NET framework allows for query language with first-class status – new C# keywords, flexible syntax, and lambda expressions for more expressive anonymous functions. The syntax of LINQ in C#, specifically, closely mimics previous query syntax suggestions [11]. When utilized with

the ADO.NET Entity Framework from earlier, a technology known as LINQ-To-Entities, the ORM provides the mapping code and a virtual ODBMS experience can be had on top of your popular RDBMS (providing, of course, there are EF .NET providers for the RDBMS).

With the adoption of some object-oriented features and the maturation of ORMs and their capabilities, RDBMS is perfectly suitable for a wide variety of application types. There are situations where an ODBMS is better suited for the task at hand, for more than a desire to ease the code burden on a developer [14]. Applications that utilize embedded DBMS's, often for on-demand or real-time applications that require easy access and make frequent changes to data, are an excellent candidate for an ODBMS. The lack of impedance when using an ODBMS greatly improves performance in execution time in such situations. Complex object relationships and specific data types, such as audio and video files, become trivial when stored in an ODBMS. As discussed before, the relational representation of multiple object relationships is verbose and difficult to manipulate and large data types like audio and video media simply lack an easy-to-utilize data type for storage.

Lastly, highly mutative data models lend themselves excellently to ODBMS use. Database schema changes and management are complex and tedious undertakings in an RDBMS, particularly when the schema is trying to accommodate changes to an object model and not a relational model. Flexibility is also a concern of the Agile software development methodology. Agile practices emphasize quick, adaptive software development, usually operating on relatively short development cycles and primarily concerned with the immediate task at hand. Again, an ODBMS that can easily handle changes in object schema would be an excellent choice for the fast-paced, highly dynamic software development cycle.

This last observation deserves further investigation. If a new application to be designed is known ahead of time to work with highly dynamic data, with changes in object shapes, interfaces, and behavior occurring regularly, can anything be done about it? Can this attribute be taken advantage of and be used as an asset of the design rather than a hindrance, dreading constant schema updates and version management? An ODBMS would help in such a situation, but an approach to such a situation would be more valuable than a specific tool for it, and ODBMS, as discussed, are nowhere near the ubiquitous nature of the RDBMS. Can something be done for such applications that must work against an RDBMS?

5.2. ADAPTIVE-OBJECT MODELING (AOM)

Of course there is. The author cannot pass on such an excellent transition. The Adaptive-Object Modeling pattern [24] handles the impedance mismatch by side-stepping it altogether. In the systems and architectures discussed thus far, the database defines the data model schema (a description of the data model – metadata) and the classes surface the entities in the schema. If the definition of an Employee changes, the classes need to be edited to accommodate the changes in the schema. Instead of defining the entities as classes, Adaptive-Object Modeling (AOM) uses classes to define the basic entity types and uses descriptions in an external source that are interpreted by the application at runtime to establish the particular configurations of types. The architecture relies heavily on a handful of established design patterns, such as Type-Objects, Strategies, and Properties [25] [24].

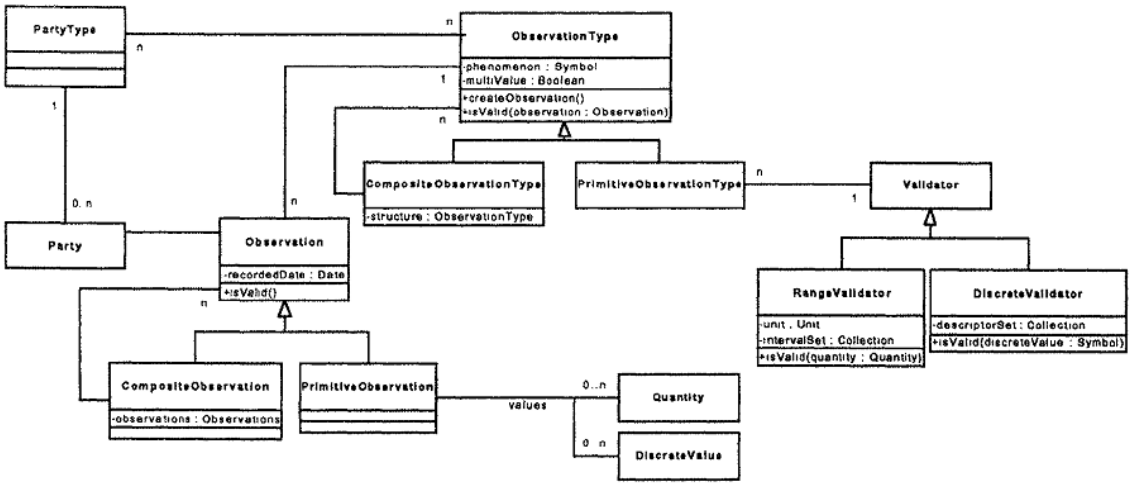


Figure 7 The AOM Pattern [24]

Static classes defined in a programming language are, almost by definition, difficult to change and manipulate. Instead of using the classes to define specific configurations of various attributes and behaviors, AOM uses the Type-Object design pattern to define the basic shapes of all possible properties and functionality and leaves their assignment to entities at runtime. An EntityType is created to explicitly define the type of entity as a class and is used for runtime instantiation when the external description files are interpreted. The descriptions are nothing more than organizations and configurations of the various EntityTypes, Operations, and Attributes. Operations and Attributes are separate classes (implementations of interfaces in the case of Operations) that define each attribute and operation available to entities across the system. The EntityType classes do not have any attributes or operations declared in the classes themselves, which would be a static coupling, but rather simply the mechanisms to handle bags of properties and methods assigned at runtime [25].

Just because attributes and operations are defined unconnected to the entities that will use them does not mean that a degree of complexity must be avoided. Through the

use of other patterns, such as the Accountability pattern, properties can still have constraints and logic governing their usage. Cardinality requirements and self-referencing and inter-entity associations can all be accounted for in this dynamic environment. EntityTypes can be designed to surface specific relationships or rules between entities. What each type CAN do is defined statically in code, what each type actually DOES is defined dynamically at runtime.

It is necessary to point out some short comings and limitations of the Adaptive-Object Modeling pattern. Many of these are commonly found in dynamic execution environments and are simply part of the cost of taking advantage of their flexibility and dynamic abilities. The rearranged architecture leaves the developer without strong-typed support at design time, including features such as IntelliSense [27] and compile-time type checking. While not debilitating by any means, it can certainly require adjustments for developers who rely heavily on such features in addition to greater exposure to tricky bugs.

Implementing dynamic functionality in a largely static OO programming language usually relies on reflection (the examination of an object's description or meta-data) a particularly expensive operation. A real-time or performance-centric application is not well-suited for a pattern like AOM because its heavy use of reflection introduces unacceptable performance degradation to the systems.

The Adaptive-Object Modeling pattern represents a drastic departure from the standard assignment of architecture responsibilities across a system's components and can be difficult to understand and follow. A thorough understanding of not only the particular application, but also AOM, is required of developers before changes can be made with confidence. This is true of any application, but is of particular importance

when working with such a pattern, and must be given full appreciation when deciding how to design or who to hire [25].

These hurdles aside, AOM offers an excellent solution for a variety of data models. Models open to user (normal or power users) input and manipulation thrive in an AOM architecture, as the models of the system can change from day to day dependent upon user editing. Businesses that involve a finite number of complex objects interrelated in multiple, hierarchical ways can also greatly increase their capabilities with software that utilizes runtime interpretation of its entities. The author has personally worked on a system that was entirely centered on a complex set of business entities and rules governing their interactions and relationships. This core business domain was not static and received continual changes and tweaks over time, due to both changing specifications and user edits. Such an environment is ideal for AOM, where the core domain is defined in an AOM architecture and peripheral, more-traditional programs or modules tap into the core as needed.

Naturally, it is certainly not applicable in all data modeling situations and only serves as another reminder to the reader that modeling an application's data, and how to go about it, is specific to each situation at hand – there is no "holy grail" of data modeling technologies or patterns.

So, how do Adaptive-Object Modeling, plumbing code, and model mappings relate to each other in terms of this paper? Recall the intention of the paper is to explore what can be done to alleviate the pain of working with an object-oriented language against a relational backend. AOM is a pattern that anticipates and takes advantage of the highly dynamic nature of its model, removing the normal headaches associated with such behavior in a standard architecture, like constant changes to schema and static class

definitions. It is simply another approach to add to the developer toolbox that can reduce development and maintenance time on the proper applications.

6. Conclusion

At the design level, impedance mismatch is a largely unavoidable aspect of any given application. However, many different mitigation strategies exist at the implementation level to reduce or remove entirely the mismatch and any associating symptoms. By taking advantage of the nature of relational databases, its similarities to objects, and knowing the predictable, consistent steps for correct translation, tools have been developed that remove or side-step this "plumbing" code from a developer's responsibilities. This allows a more productive, efficient, and enjoyable development experience for both developer and management.

References

[1] A. Aarsten, D. Brugali, and G. Menga. Patterns for three-tier client/server applications. In *Proc. of PLop*, volume 96. Citeseer.

[2] A. Adya, J.A. Blakeley, S. Melnik, and S. Muralidhar. Anatomy of the ado. net entity framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, page 888. ACM, 2007.

[3] S. Alagic and P.A. Bernstein. A model theory for generic schema management. *Lecture Notes in Computer Science*, pages 228–246, 2002.

[4] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, volume 57. Citeseer, 1989.

[5] C. Bauer and G. King. *Hibernate in action*. Manning, 2004.

[6] P.A. Bernstein. Applying model management to classical meta data problems. In *Conference on Innovative Data Systems Research (CIDR)*, pages 209–220. Citeseer, 2003.

[7] P.A. Bernstein, A.Y. Halevy, and R.A. Pottinger. A vision for management of complex models. *ACM Sigmod Record*, 29(4):55–63, 2000.

[8] P.A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, page 12. ACM, 2007.

[9] G. Cernosek. Next-generation model-driven development. *IBM White Paper*, 2004.

[10] W.R. Cook and A.H. Ibrahim. Integrating programming languages and databases: What is the problem. *ODBMS. ORG, Expert Article*, 2006.

[11] W.R. Cook and C. Rosenberger. Native Queries for Persistent Objects A Design White Paper. *Dr. Dobbs Journal*, 2006.

[12] A.M. Geoffrion. An introduction to structured modeling. *Management Science*, 33(5):547–588, 1987.

[13] R. Grehan. ODBMS for RDBMS Users. *ODBMS. ORG: Object Database Management Systems*, 5.

[14] R. Grehan. When to Use an ODBMS. *ODBMS. ORG: Object Database Management Systems*, 5.

[15] R.A. Heubner, G. Oancea, R.P. Donald, and J.E. Coleman. Object model mapping and runtime engine for employing relational database with object oriented software, August 8 2000. US Patent 6,101,502.

[16] R. Hirschfeld. Three-Tier Distribution Architecture. *Collected papers from the PLoP*, 96:97–07.

[17] P.C. Kanellakis. Elements of relational database theory. *Handbook of theoretical computer science*, pages 1075–1144, 1990.

[18] N. Leavitt. Whatever Happened to Object-Oriented Databases? *IEEE Computer*, 33(8):16–19, 2000.

[19] E. Marcos, B. Vela, and J.M. Cavero. A methodological approach for object-relational database design using UML. *Software and Systems Modeling*, 2(1):59–72, 2003.

[20] L.E. McKenzie and R.T. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.

[21] E.J. O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356. ACM, 2008.

[22] J. Wetherbee. Integrating relational databases in an object oriented environment, August 10 1999. US Patent 5,937,409.

[23] wikipedia.org. ADO.NET. http://en.wikipedia.org/wiki/ADO.NET.

[24] J.W. Yoder, F. Balaguer, and R. Johnson. Architecture and design of adaptive object-models. *ACM Sigplan Notices*, 36(12):50–60, 2001.

[25] J.W. Yoder and R. Johnson. The adaptive object-model architectural style. *Urbana*, 51:61801.

[26] J.W. Yoder, R.E. Johnson, and Q.D. Wilson. Connecting business objects to relational databases. *Urbana*, 51:61801.

[27] wikipedia.org. IntelliSense. http://en.wikipedia.org/wiki/IntelliSense.

Vita

Kenneth Walter Fiduk, III was born in Dallas, Texas on August 31, 1980, the son of Rose Novak Fiduk and Kenneth Walter Fiduk, Jr. Despite being only three at the time, Kenneth did everything in his power to relocate his family to Austin, TX, where he has since resided. After completing his work at Westwood High School, Austin, TX, in 1998, he entered the University of Texas at Austin where he first studied Electrical Engineering before turning to his love at heart, Computer Science. He received the degree of Bachelors of Science in Computer Science, with High Honors and Department Honors, in May, 2003. During the following five years, he was largely employed as a software developer at e-MDs, a medical software company, before branching out to other local opportunities. In January, 2008 he entered the Graduate School of the Cockrell School of Engineering at the University of Texas at Austin.

Permanent Address: 7607 Bellflower Cove
Austin, Texas 78759

This report was typed by the author.